Verilog-HDL入門

2014年12月1日 修正： 2010年8月3日 公開

内田智久 E-sys, IPNS, KEK

はじめに

回路設計未経験者向けに必要最低限のVerilog-HDL文法を解説した入門書です。

専門家向けに書かれた市販書籍は情報が多すぎるため、回路設計初心者からみると最低限何をどのように使えば良いのか分かりません。これは、対象読者が論理回路設計経験（出来る事）を前提に書かれてているからです。

この文書の目的は“とにかくHDLで回路を表現できる事”であり、作業効率を上げるための便利な記述やエレガントな記述などは解説しません。この文章を理解した後は市販の書籍を読んで自分にあった記述方法を習得してください。

予備知識

- ブロック図と回路図とは何かを知っている
- 階層構造回路の概念
- 基本4要素AND, OR, INVゲートおよびD-FFの動作を理解している
- 同期回路設計を知っている
- ステートマシンを知っている

この文書では設計方法は同期回路設計、シミュレーションは論理シミュレーションの2つを使用する事を前提に話を進めます。

目次

        - シフトレジスタ
        - ステートマシン

4. シミュレーション記述

    - シミュレーションに必要な物
    - テストベンチ
    - シミュレーション特有の記述
    - 4.1 カウンターのシミュレーション
        - テストする回路
        - テストベンチ
        - シミュレーション結果

参考文献

以下の文献を参考にさせていただきました。

- 小林　優、“入門Verilog-HDL記述”、CQ出版
- 小林　優、デザインウェーブ付録“はじめてでも使える　HDL文法ガイド”、CQ出版
- 枝　均、“Verilog-HDLによるテストベンチ”、テクノプレス
- 安岡 貴志、“デジタルデザインテクノロジ10月増刊号 Verilog HDL & VHDL テストベンチ記述の初歩”、CQ出版

免責事項等

技術教育のページへ戻る

1. Verilog-HDLの構造

Verilog-HDLの概要を説明します。

module

Verilog-HDLではテキストファイルにより回路情報を記述します。また、回路ブロックをモジュール(module)と言う単位で表します。Verilog-HDLはモジュールの集合体です。モジュールの区別はモジュール名と言う回路機能名で行います。

- moduleの分割は設計者が自由に決める事が出来ます
- moduleは機能単位で分割する事が多いです
- moduleは階層構造を取る事が出来ます
- module名は設計者が自由に名付ける事が出来ます。
- moduleはテキストファイルで記述されます
- 通常の拡張子は.vです
- 1ファイルで1回路ブロック（モジュール）とすることを勧めます
- 文法上は1ファイル中に複数の回路ブロックを記述しても良い事になっていますが、分かりやすく1ファイルで1回路ブロックを記述すると良いと思います。
- 半角英数字記号が使用できます
- コメント内では全角文字も使用できることになっていますが、使用すると挙動がおかしくなるツールがあるので使用する場合は十分に注意してください。

moduleとは回路を記述したファイルです。ある部品の回路（動作）をHDLで記述したまとまりと思ってください。

階層構造について

moduleは階層構造をとる事が出来るので、階層構造を持つ回路図と同じように扱う事が出来ます。

あるモジュールの中から他モジュールの呼び出し。複数のモジュールから同じモジュールの呼び出し（コピー、使いまわし）ができます。各モジュールは他のモジュール内に組み込まれたときに初めて回路として定義されます（最上位のトップモジュールは除く）。

例えば、module-Aの中でmodule-Bを2つ使用したとします。下の様にmodule-Aの中でmodule-Bが2箇所で組み込まれているとします。

```
module-A -+- module-B
          |
          +- module-B
```

この場合、2つのmodule-Bは独立な回路として動作します。この2つのmodule-Bが区別できないと困りますので実際に組み込む時は下の様に回路番号をつけます。PCBボードのリファレンス番号（部品番号）と同じです。

```
module-A(TOP) -+- module-B(B1)
               |
               +- module-B(B2)
```

カッコの中がモジュール番号です。このモジュール番号をVerilog-HDLでは**インスタンス名**と呼びます。

まとめると、モジュールは2つの名前を持ちます。それは、モジュール名とインスタンス名です。

名前について

Verilog-HDL内ではモジュール名、インスタンス名、信号名など様々な名前を使用します。名前の付け方には規則があります。

識別子 :名前を総称して識別子と呼びます。通常の識別子には下の様な規則があります。

- 最初の文字は半角英字またはアンダースコア
- 識別子は半角英字、数字、アンダースコアで構成
- 大文字と小文字は区別される
- キーワードは使用できない

キーワードとは予約語の事でVerilog-HDLの文法制御の為に予約されているものです。例えば、assign, moduleなど。

数値

デジタル回路は2進演算回路ですから数字を扱う事が多いです。様々な基数表現を用いる事が出来ます。

```
[ビット幅]'[基数][値]
```

値の表現は、最初にビット幅（正整数）、シングルクォーテーション、基数、値の順に記述します。

最初にビット幅を書きます。ソフトウエアと異なりハードウエアでは値を表現する為に使用する信号線数（ビット幅）を意識しなければいけません。

次にシングルクォーテーションを入れます。

基数は下の記号を使って表します。

記号	基数
bまたはB	2
oまたはO	8
dまたはD	10
hまたはH	16

例えば、値128を8bit幅、16進で使用する場合は下のようになります。

```
8'h80
```

値128を8bit幅、10進で使用する場合は下のようになります。

```
8'd128
```

ある部分は省略する事が出来ますが、初めは丁寧に一つずつ記述する事をお勧めしておきます。

特別な数値

シミュレーション用に以下の数値が用意されています。

記号	基数
zまたはZ	ハイ・インピーダンス
xまたはX	不定

ハイ・インピーダンス

別名スリーステート(3-state)。ドライブされていない信号状態の事です。

目次へ戻る

技術教育のページへ戻る

2. モジュール構造

Velirog-HDLコードは以下の特徴を持ちます

- テキストファイルで記述される
- 拡張子は通常.vを使用
- 一つ以上のモジュールで構成される

1つのテキストファイルに複数モジュールを記述する事が出来ますが、1モジュールを1ファイルで記述する事が多いです。

モジュール構造

moduleから始まりendmoduleまでが一つのモジュール。この間に回路を記述する。

```
module モジュール名 （ポートリスト）;

    ポート宣言：モジュールに入出力される信号の入出力を定義

    パラメータ宣言：
    内部信号の属性：レジスタ出力か否か

    回路記述
        下位モジュールの呼び出し
        assign文、always文など

endmodule
```

2入力XORを記述したMY_XOR2モジュールの例を挙げます。

```
/* モジュール例として2入力XORを書きました
   この行もコメントになります。
*/

module MY_XOR2(
    input  IN_A    , // 入力A
    input  IN_B    , // 入力B
    output O         // 出力O
);

//---------------------------------------------------------
// XOR
//---------------------------------------------------------
    wire IA; // 組み合わせ回路の出力はwireで定義する
    wire IB;

    assign  IA =  IN_A & ~IN_B;
    assign  IB = ~IN_A &  IN_B;

    assign  O = IA | IB;

endmodule
```

上から順に説明します。

コメント

先ずはコメントについて説明します。

コメントを入れる事が出来ます。コメントを表す表現は下の2種類あります。

- /*から*/まで。複数行にわたる事が出来ます。
- //から行末まで

日本語使用について

コメント以外の場所に全角スペースなどの全角文字を入れる事はツール誤動作の原因になりますので注意して下さい。コメント内では日本語も使えますが、以

前はコメントに日本語を入れると開発ツールの動作が異常になる事がありました。どう考えても記述と異なる動作をするので日本語コメントを外したら正常に動作したという経験があります。注意して使ってください。

モジュール名とポートリスト

```
module MY_XOR2(
    input  IN_A    , // 入力A
    input  IN_B    , // 入力B
    output O         // 出力O
);
```

moduleがモジュールの始まりです。スペースを空けてMY_XOR2がモジュール名です。その隣の(から);までがポートリストです。

ポートリストとは入出力信号のリストです。このカッコで挟まれる間に、このモジュールMY_XOR2に入出力される信号の入出力定義と信号名を記述します。ポートリスト最初の信号を例に説明します。

```
    input  IN_A    , // 入力A
```

この例では、inputが入出力定義を表し、IN_Aが信号名を表しています。ポートリストでは左に入出力定義を記述し空白を入れて右側に信号名を記述します。各信号はカンマで区切ります。

入出力定義は以下の3種類あります。

キーワード	意味
input	入力
output	出力
inout	入出力（双方向）

FPGA設計時の注意

FPGA内部で入出力信号は使用できません。仮に使用した場合は開発ツールが自動的に入力と出力の独立した信号へ変換します。もちろんFPGA外部とのI/F信号では使用する事が出来ます。

信号名は設計者が自由につける事が出来ますが識別子規則に従う必要があります。このモジュールは3つの信号：IN_A, IN_B, Oを持っている事が分かります。各信号はカンマで仕切ります。各信号の右にある//はコメントの開始を意味しています。//から行末まではコメントと解釈されます。コメントはこの他に/*から*/で囲む事でも表す事が出来ます。

Verilog-HDLはセミコロンで分かれている単位で解釈します。従って、一文の最後には必ずセミコロンをつけなければいけません。セミコロンまでを一文と認識するので途中にタブやスペース改行を任意に入れる事が出来ます。この例の様にコメントも入れる事が出来ます。自分が見やすいように記述して下さい。

例えば、下の様に書くこともできます。

```
module MY_XOR2(input IN_A, input IN_B, output O);
```

信号の属性

```
    wire IA; // 組み合わせ回路の出力はwireで定義する
    wire IB;
```

信号の属性を定義します。

下2つの属性があります。

属性	意味
wire	組み合わせ回路出力の信号
reg	フリップフロップ（記憶素子）出力の信号

最初は記憶素子出力の信号は**reg**、そうでない信号は**wire**で宣言すると覚えてください。

属性の宣言は信号を初めて使用するより前（上の行）で行う必要があります。例えば、下は正しい例です。Bの宣言位置に注目してください。

```
wire B;
wire C;

assign C = B;
assign B = A;
```

下の記述はエラーになります。

```
wire C;
assign C = B;

wire B;
assign B = A;
```

上の例の場合、

```
assign C = B;
```

でBを使用していますが、Bの宣言が後（下の行）にあるためにエラーになります。信号を初めて使用するより前（上の行）で宣言してください。

内部信号については全て宣言しなければいけません。内部信号とはポートリストに表れない信号の事です。ポートリストに表れる入力信号については宣言してはいけません。実はこの例で用いたポートリストの書き方は下の記述と同じ意味になります。

```
module MY_XOR2(
    input  wire IN_A    , // 入力A
    input  wire IN_B    , // 入力B
    output wire O         // 出力O
);
```

違いは入出力定義と信号名の間に信号の属性が追加されていることです。多くの場合ポートリストの信号の属性はwireであるため記述の省略が許されていたのです。従って、ポートリストの信号は既にwireで宣言されていると見なされるため下部でwire宣言すると重複宣言となりエラーとなります。

【補足】

シミュレーションを行う時にテストベンチを用いますが、ベンチから与える信号はregで定義します。これは、記憶素子に直接値を代入していると考えてください。通常はテストする回路へ信号を与える為にある信号の変化点のみ時間と値を設定するので記憶素子として定義しないと指定した瞬間しか値を指定できません。

回路記述

ここからendmoduleまでの部分に回路を記述します。

回路の記述法を後の章で説明します。この文書ではassign文とalways文の2種類を説明します。シミュレーション記述ではinitial文を説明します。

他の表現方法もありますが、便利な書き方、作業効率を上げる書き方、エレガントな恰好が良い書き方に属する記述法とみなし説明しません。先の3つの文を学習した後に各自書籍等で習得してください。

先ずは先にあげた3つを習得しましょう。ちなみに、私はコードのほぼ全てをこの3つの文を使って書いています。

```
    assign  IA =  IN_A & ~IN_B;
    assign  IB = ~IN_A &  IN_B;

    assign  O = IA | IB;

endmodule
```

下位モジュールの組み込み

モジュールは階層構造を取る事が出来ます。

例えば、ここで使用したMY_XOR2を親モジュールTOPに組み入れ3入力XORとして使用する例を見てみます。

```
/* MY_XOR2を2つ組み込んだ例です */

module TOP(
    input  IN_A    , // in    : Input A
    input  IN_B    , // in    : Input B
    input  IN_C    , // in    : Input C
    output O         // out   : Output
);

    wire            Z     ;

    MY_XOR2 U1(
        .IN_A (IN_A),
        .IN_B (IN_B),
        .O    (Z)
    );

    MY_XOR2 U2(
        .IN_A (IN_C),
        .IN_B (Z),
        .O    (O)
    );

endmodule
```

endmoduleの上の部分でMY_XOR2を組んでいます。

【補足】下位モジュールのVerilog-HDLファイルの指定

下位モジュールMY_XOR2を組み込む場合はMY_XOR2の回路が記述されたVerilog-HDLファイルMY_XOR2.vを使用します。このMY_XOR2.vの読み込ませ方は使用する方法は**include**文を用いて明示的に指定する方法と**開発ツール**を用いて指定する方法です。現在は**include**文で指定する方法より開発ツールにより指定する方式を使用する事が多くなっています。ここでは、開発ツールで指定する事を仮定して進めます。

```
    MY_XOR2 U1(
        .IN_A (IN_A),
        .IN_B (IN_B),
        .O    (Z)
    );
```

MY_XOR2はモジュール名（回路の名前）、**U1**がインスタンス名（モジュール番号）です。その後ろにあるのがポートリストでTOPモジュールの信号とMY_XOR2モジュールの信号との接続関係を定義します。この例では、MY_XOR2モジュールのIN_AをTOPモジュールのIN_Aに、MY_XOR2モジュールのIN_BをTOPモジュールのIN_Bに、MY_XOR2モジュールのOをTOPモジュールのZに接続しています。下位モジュールの信号は信号名の左にドットを付けて表します。その右に括弧で囲んだ接続する信号名、ここではTOPモジュールの信号名、を記述します。下位モジュールと上位モジュールの信号名は異なって構いません。階層が異なれば別信号と認識します。

```
    MY_XOR2 U2(
        .IN_A (IN_C),
        .IN_B (Z),
        .O    (O)
    );
```

同様に次をみると、U1の出力がIN_Bに接続され、出力がTOPモジュールの出力Oに接続されています。

インスタンス名

同一モジュール（階層）内で同じインスタンス名を付ける事はできません。モジュール（階層）が異なれば使用できます。

目次へ戻る

技術教育のページへ戻る

3. 回路記述

3.1. 組み合わせ回路

assign文は組み合わせ回路を記述するときに使用します。

組み合わせ回路を表現するためにはANDゲート、ORゲート、NOTゲートを使用します。先ずは、これらの記述方法を見てみましょう。

```
    assign  出力信号 = 論理式;
```

assign、出力信号名、等号、論理式、最後にセミコロンで終了します。複数行にわたって記述する事が出来ます。

再び2入力XORの例をあげます。

```
    assign  O = (IN_A & ~IN_B) | (~IN_A & IN_B);
```

ここで、Oは出力信号、(IN_A & ~IN_B) | (~IN_A & IN_B)の部分が論理式です。算術演算の順序等を指定するためにカッコを使用する事が出来ます。先に複数行で記述しても良いと書きましたが、以下の様に記述しても構いません。

```
    assign  O = (IN_A & ~IN_B) |
                (~IN_A & IN_B);
```

分かりやすく書く工夫をしてみてください。

論理式で使用する事が出来る演算子の中から使用頻度が高い物を選んで説明します。

論理演算

演算子	意味
&	AND（論理積）
|	OR（論理和）
~	NOT（反転）
^	XOR（排他的論理和）

先のXORの例を再び見てみましょう。

```
    assign  O = (IN_A & ~IN_B) | (~IN_A & IN_B);
```

IN_AとIN_BのNOTとのANDとIN_Aの反転とIN_BのANDのORを記述しています。カッコの中の演算が優先されます。演算子には優先順位がありますのでカッコが不要な事がありますが、慣れるまでは分かりやすさを優先して大袈裟にカッコを付ける事を勧めておきます。

【補足】演算子の優先順位

高いものから、NOT, AND, XOR, ORの順です。

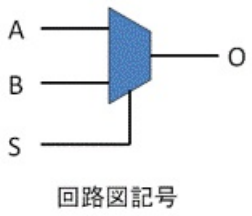
先の演算子の表中にXORがありますので、先の例は以下のように簡単に書く事が出来ます。

```
    assign  O = IN_A ^ IN_B;
```

3項演算子

セレクター、マルチプレクサを記述する時に便利な3項演算子を説明します。

2つの入力A,Bを選択信号Sにより選択するセレクター回路を考えます。



回路図記号

入出力関係

A	B	S	O
L	X	L	L
H	X	L	H
X	L	H	L
X	H	H	H

XはDon't careで何でも良いと言う意味

この回路をVerilog-HDLで記述するには下の様に書きます。

```
    assign  O = (S ? B : A);
```

見やすくするために括弧を入れましたが、無くても大丈夫です。

この一行でセレクタを表現する事が出来ます。次に説明するマルチビット表現と組み合わせてデータバス選択回路として使用されます。

【補足】3項演算子の優先順位

演算子の中で最も低いです。

マルチビット表現

信号は識別子を使用しますが、マルチビット表現を使用する事が出来ます。デジタル回路では値を表現するために複数本の信号を一つの値をとして使用する事があります。例えばアドレスやデータなど。32bitデータは32本で意味をもちます。このような信号を一つずつ定義する事は大変ですし分かりにくくなるので下の様に一つの識別子で表現する事が出来ます。

次にマルチビット表現を使用したモジュール例を見てみましょう。

```
/* マルチビット表現の例です */

module DECODER(
    ADDRESS , // in    : Input  , Address[7:0]
    CE        // out   : Output , Chip enable[1:0]
);

// ----- input/ output -----
    input   [7:0]   ADDRESS   ;
    output  [1:0]   CE        ;

//-------------------------------------------------------
// Address decoder
//-------------------------------------------------------
    wire   [1:0]    CE    ;  // 組み合わせ回路の出力はwireで定義する

    assign  CE[0] = (ADDRESS[7:0]==8'h0);
    assign  CE[1] = (ADDRESS[7:0]==8'h1);

endmodule
```

ポートリスト

マルチビットになってもポートリストは変更ありません。この部分はあくまで信号名を記述します。

ポート宣言

```
    input   [7:0]   ADDRESS   ;
    output  [1:0]   CE        ;
```

入出力定義と信号名の間に[u:l]形式でビット範囲を指定します。[u:l]はビット範囲を表しています。左側uをMSBにコロンで区切って右側lにLSBを書きます。LSB, MSBの値は任意です。

補足

[l:u]の様に左にLSBを書く人もいますが、左をMSBにする人が多いと思います。

ここでは**ADDRESS**はLSBが0, MSBが7のビット幅8の入力信号、**CE**はLSBが0, MSBが1のビット幅2の出力信号を宣言しています。

内部信号の属性

```
    wire   [1:0]    CE    ;  // 組み合わせ回路の出力はwireで定義する
```

ポート宣言と同じように信号の属性と信号名の間に[u:l]形式でビット範囲を指定します。

回路の記述

```
    assign  CE[0] = (ADDRESS[7:0]==8'h0);
    assign  CE[1] = (ADDRESS[7:0]==8'h1);
```

マルチビット表現の信号は数値として扱う事が多いため、論理演算以外にも上のように等号や不等号など条件抽出を使用する事があります。算術演算も使用する事が出来ます。

条件演算子: 使用できる条件演算子

演算子	意味
==	等しい
!=	等しくない
>	(左辺が)大きい
<	(左辺が)小さい
>=	(左辺が)大きいまたは等しい
<=	(左辺が)小さいまたは等しい

条件が成立した時に**1**になります。条件が成立しない時は0になります。 `assign CE[0] = (ADDRESS[7:0]==8'h0);`

この場合、**ADDRESS[7:0]**が**0**の時のみ**CE[0]**が**1**になります。

定義されているLSBとMSBの間の任意の範囲を指定できます。例えば、**ADDRESS**のビット1から6までを抜き出す場合は下のように記述します。

```
    assign  CE[0] = (ADDRESS[6:1]==6'h0);
```

マルチビットとして定義されている信号の中から一つの信号を抜き出して使用するときは下のように書きます。

```
    assign  CE[0] = ADDRESS[0];
```

または

```
    assign  CE[0] = (ADDRESS[0]==1'b1);
```

この場合、ビット0を抜き出しています。

算術演算

演算子	意味
+	加算
-	減算

例えば、信号A[7:0]に1を加えて信号O[7:0]に出力する時は下の様に書きます。

```
    wire   [7:0]  O;

    assign  O[7:0] = A[7:0] + 8'd1;
```

桁上がりについて

上の例で9ビット以上の桁上がりは無視されます。例えば、A[7:0]=0xFFに8'd1を加えたらO[7:0]=0となります。桁上がりがあれば0x100となるのですが9ビット目は無いので8ビット以下の桁のみ正しく演算されます。

積算、除算、剰余

上記以外にも積算、除算、剰余演算子も定義されていますが回路記述に使用する事はほとんどありません。これらの回路規模は大きく、実現方法も数多く存在します。多数の実装方法の中から適した回路を指定するためには、ゲートレベルを想像できる程度に分解した記述をするかゲートレベルで回路が指定されているライブラリを使用する必要があります。

シミュレーション

記述を回路へ変換する必要が無いシミュレーションのテストベンチでは積算、除算、剰余演算子を使用した方が高速に動作しますので必要であれば積極的に

使ってください。

ビット結合

マルチビット表現された複数の信号まとめて新しいマルチビット表現した信号を定義したい時があります。

例えは、信号Aが[7:0]、信号B[15:0]で定義されている信号を合わせて一つの信号C[23:0]を定義したい時を考えてみます。C[15:0]にB[15:0]を接続し、C[23:16]にA[7:0]を接続します。

下の様に一本ずつ定義すれば問題ない事はわかると思います。

```
    wire  [23:0]  C;

    assign  C[0] = B[0];
    assign  C[1] = B[1];
      :       :      :
    assign  C[15] = B[15];
    assign  C[16] = A[0];
      :       :      :
    assign  C[23] = A[7];
```

しかし、これは大変です。そこで、下の様に書ける事に気がつくかもしれません。

```
    wire  [23:0]  C;

    assign  C[ 15:0] = B[15:0];
    assign  C[23:16] = A[7:0];
```

これでも良いのですが、一文で書く方法が文法で定義されています。下の様に記述します。

```
    wire  [23:0]  C;

    assign  C[23:0] = {A[7:0],B[15:0]};
```

ここでは右辺で使用しましたが、左辺（出力側）でも使用する事が出来ます。

```
    wire  [15:0]  D;
    wire  [7:0]   E;

    assign  {D[15:0],E[7:0]} = {A[7:0],B[15:0]};
```

3.2 記憶素子を使った回路

順序回路記述の前に記憶素子を使用した回路記述を説明します。

ここでは、記憶素子としてDタイプ・フリップフロップ(DFF)を想定します。他のタイプの素子については参考文献を参照してください。

DFFは単ゲートよりも機能が多いのです。以下の機能を部分的または全て持っています。

- リセット(非同期または同期)
- プリセット(非同期または同期)
- クロックイネーブル

Xilinx社は正論理信号の使用を奨励していますので、ここでは正論理の信号を使用した記述から始めます。

D-type Flip Flop (DFF)

リセットやクロックイネーブルなどが無いDFF単体の基本的な記述です。

クロック名をCLK、入力信号をD、出力信号をQとします。

```
   reg      Q;

   always @(posedge CLK)begin
      Q <= D;
   end
```

この文の意味を説明します。

always文

キーワードalwaysに続けて書かれている"@()"の中の条件文が成立する時にキーワードbeginとendの間にある文が動作します。この場合posedge CLKが動作条件でです。posedgeはキーワードで立ち上がりエッヂを意味しています。従って、この回路はクロック入力CLKの立ち上がり時に毎回動作します。逆の言い方をすれば、クロック入力CLKの立ち上がり時だけ動作します。それ以外の時は動作しません。動作しないとは状態を保持したままになると言う意味です。

まとめるとCLKの立ち上がりエッヂの時のみ入力DをQに出力し、他の時は状態を保持します。DFFの動作になります。

動作記述の文中等号が=でなく<=である事に気がついたでしょうか。これは大小比較ではありません。実は=と<=の両方を使う事が出来ます。そして意味が違います。ここではDFFとして使用する場合は等号=ではなく<=を使うと覚えてください。

always文中の=と<=の違い

=を使用した場合、always文中の式を全て評価した最終結果を採用します。**<=**を使用した場合、セミコロンで評価を終わらせます。この違いは一つの**always**文中に複数の式を記述した場合に起こりやすいです。

always文とDFF

always文はDFF以外の回路も表現する事が出来ます。例えば、組み合わせ回路も表現できます。しかし、always文は書き方が少し複雑であり生成される回路を意識する事が難いです。最初は組み合わせ回路はassign文で記述し、DFF出力はalways文のひな形を使用すると決めて設計する方が無難です。慣れてきたら参考文献を参考にして自分に合った記述を身につけてください。

beginとend

文の構造を明示するときに使用するキーワードです。C言語のカッコ{}に似ています。

Verilog-HDLの場合、セミコロンまで一つの文と解釈するので上の例は下の様に省略する事が可能です。

```
   always @(posedge CLK)
      Q <= D;
```

このように省略する事が出来ますが構造を意識して記述する事を重視するとbegin, endの使用を勧めます。always文の中の式が複数になる場合はbegin, endが必要です。

posedgeとnegedge

posedge以外にnegedgeもあります。クロックの立下りで動作するDFFを使用する場合は下の様に記述してください。

```
   always @(negedge CLK)begin
      Q <= D;
   end
```

HDLに慣れるまではDFFはこのように書くと覚えてください。慣れてきたら文の意味を再度確認してください。実はalways文の意味と動作を厳密に結びつけようとすると解釈に疑問だでる箇所があります。私自身質問されると困る文が存在します。

同期リセット付きDFF

クロック名をCLK、リセット入力をRESET、入力信号をD、出力信号をQとします。

同期リセット付きDFFの記述は下のようになります。

```
reg      Q;

  always @(posedge CLK)begin
      if(RESET) begin
          Q <= 1'b0;
      end else begin
          Q <= D;
     end
  end
```

if文が追加されました。ifに続く()の中の条件が成立（真の時）した時はifに続く文を実行します。このif文の条件文の意味は''RESET==1'b1''と同じ意味です。RESETが1の時下の文が実行されます。

```
          Q <= 1'b0;
```

条件が成立しないときはelseに続く下の文を実行します。

```
            Q <= D;
```

動作をまとめるとクロック入力CLKの立ち上がりエッヂでRESETが1の時は0を出力し、立ち上がりエッヂでRESETが0の時はDを出力する回路になります。同期リセット付きDFF回路です。

同期リセットおよびクロック・イネーブル付きDFF

クロック名をCLK、リセット入力をRESET、クロック・イネーブルをCE、入力信号をD、出力信号をQとします。

同期リセット付きおよびクロック・イネーブル付きDFFの記述は下のようになります。

```
   reg      Q;

   always @(posedge CLK)begin
       if(RESET) begin
           Q <= 1'b0;
       end else begin
           if(CE)begin
               Q <= D;
           end
      end
   end
```

先の例のelse文の中に下の部分が追加されました。

```
           if(CE)begin
               Q <= D;
           end
```

この文がelseの中に追加されているのでRESET=0の時にこのif文が動作することが分かります。if文の中の条件式がCE==1ですから、RESET=0かつCE=1の時のみ入力信号を出力する下の文が動作します。

```
               Q <= D;
```

CE=0の時の記述がありませんが、CE=0の時の動作はどうなるのでしょうか？記述が無い場合は状態を保持する決まりになっています。つまり、この記述ではCE=0の時は停止状態で状態を保持します。

実際の記述

上の例は全て入力をそのまま出力する物でしたが、右辺に組み合わせ回路を記述する事が出来ます。

例えば、組み合わせ回路で登場した2入力XORの出力にDFFを挿入した回路は以下のように記述する事が出来ます。

```
module MY_XOR2_WDFF(
    input  CLK     , // in    : Input, System clock
    input  RESET   , // in    : Input, System reset
    input  CE      , // in    : Input, Clock enable
    input  IN_A    , // in    : Input
    input  IN_B    , // in    : Input
    output O         // out   : Output
);

   always @(posedge CLK)begin
       if(RESET) begin
           Q <= 1'b0;
       end else begin
           if(CE)begin
               Q <= (IN_A & ~IN_B) | (~IN_A & IN_B);
           end
      end
   end

   assign O = Q;

endmodule
```

3.3 順序回路

記憶素子を用いた代表的な下の回路を紹介します。

- カウンター
- シフトレジスタ

カウンター 順序回路の代表です。頻繁に使用します。

下にリセット+クロックイネーブル付き8bitカウンタを示します。

```
    reg     [7:0]   Q           ;

    always@ (posedge CLK) begin
        if(RST)begin
            Q[7:0]  <= 8'd0;
        end else begin
            if(CE)begin
                Q[7:0]  <= Q[7:0] + 8'd1;
            end
        end
    end
```

基本的にはDFFの記述と同じです。算術演算+が登場している事と自分自身の出力が入力に戻ってきている事が異なります。

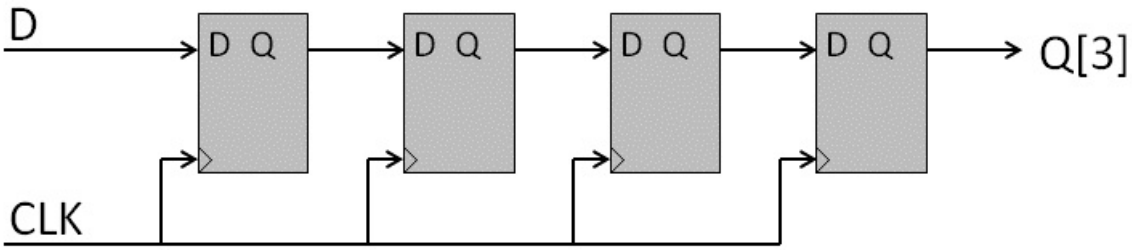
コードを読みやすくするためにキャリーを意図的に信号名を変えて表現したい時があります。その時は下の様に書きます。

```
    reg     [6:0]   Q           ;
    reg             carry       ;

    always@ (posedge CLK) begin
        if(RST)begin
            {carry,Q[6:0]}  <= 8'd0;
        end else begin
            if(CE)begin
                {carry,Q[6:0]}  <= {carry,Q[6:0]} + 8'd1;
            end
        end
    end
```

シフトレジスタ

シフトレジスタは複数のDFFを直列に接続した回路です。



この回路を素直にHDLへ変換すると下の様になります。

```
    reg     [3:0]   Q           ;

    always@ (posedge CLK) begin
        Q[0]  <= D;
    end

    always@ (posedge CLK) begin
        Q[1]  <= Q[0];
    end

    always@ (posedge CLK) begin
        Q[2]  <= Q[1];
    end

    always@ (posedge CLK) begin
        Q[3]  <= Q[2];
    end
```

これは正しいですが読みにくいです。読みやすくするために、下の様にalways文一つに入れる事が出来ます。

```
    reg     [3:0]   Q           ;

    always@ (posedge CLK) begin
        Q[0]  <= D;
        Q[1]  <= Q[0];
        Q[2]  <= Q[1];
        Q[3]  <= Q[2];
    end
```

この文を注意深く見るとマルチビット表現を使用すると下の様に簡単に書ける事が分かります。

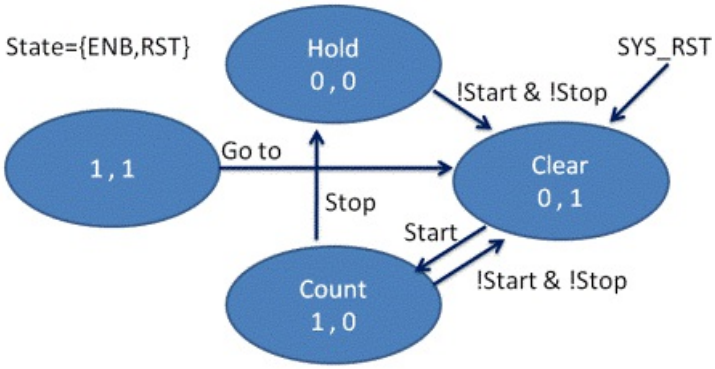
```
    reg     [3:0]   Q           ;

    always@ (posedge CLK) begin
        Q[3:0]  <= {Q[2:0],D};
    end
```

これが代表的なシフトレジスタの記述法です。

ステートマシン 制御回路の設計でステートマシンは便利な記述です。

例えば下の状態遷移図を持つステートマシンの記述について考えてみます。



動作を簡単にせつめいします。丸の中の数字はSate（出力状態）を表しています。システムリセットが入力されるとClear stateになりENB=L, RST=Hになります。Start=HでCount state（ENB=H, RST=L）へ遷移します。さらにその後Stop=Hが入力されるとHold sate(ENB=L, RST=L)になります。Start=LおよびStop=Lになるとと Clear stateへ戻ります。

これをVerilog-HDLで記述する方法は幾つかありますが、ここではcase文を使用して記述してみます。

```
    wire          RST     ;
    wire          ENB     ;
    reg   [1:0]   State   ;

    always@ (posedge CLK) begin
        if(SYS_RST)begin
            State[1:0] <= 2'b01;
        end else begin
            case(State[1:0])
                2'b01:begin  Clear
                    if(Start)begin
                        State[1:0] <= 2'b10;
                    end else begin
                        State[1:0] <= 2'b01;
                    end
                end

                2'b10:begin // Count
                    if(!Start & !Stop)begin
                        State[1:0] <= 2'b01;
                    end else if(Stop)begin
                        State[1:0] <= 2'b00;
                    end else begin
                        State[1:0] <= 2'b10;
                    end
                end

                2'b00:begin // Hold
                    if(!Start & !Stop)begin
                        State[1:0] <= 2'b01;
                    end else begin
                        State[1:0] <= 2'b00;
                    end
                end
                default:begin
                    State[1:0] <= 2'b01;
                end
            endcase
        end
    end

    assign {ENB,RST} = State[1:0];
```

目次へ戻る

技術教育のページへ戻る

4. シミュレーション記述

設計した回路が正しく動作するかの検証の為に実装する前にコンピュータ上でシミュレータを動作させてシミュレーションを行います。

シミュレーションは大別して論理シミュレーションと遅延シミュレーションがあります。論理シミュレーションは回路の動作遅延を考慮せずに（遅延を無限小として）論理動作のみ検証する方法です。遅延シミュレーションは回路の遅延情報用いてシミュレーションする方法です。

ここでは論理シミュレーションのみ説明します。

遅延シミュレーション FPGA設計をする場合、遅延シミュレーションを行う事は少ないです。シミュレーションを行う事で動作理解を深める事が出来ますので出来るだけシミュレーションするようにしてください。

たまに、シミュレーションで動くのに実機で動作しないと悩んでいる人が居ますがシミュレーションで動くのは当たり前です。シミュレーション環境を作るのは設計者であり、その環境で動くように設計した回路ですからシミュレータ上では必ず動きます。問題が発生するとき、いわゆるバグは自分が想定していない状況（信号の振る舞い）だから起こるのです。ですから、実機での入力信号を良く知った上でシミュレーションで想定できる限りの様々な条件でテストすることが重要です。

シミュレーションに必要な物

シミュレーションを行うためには設計したVerolog-HDLファイル(DUT)の他に下の物が必要です。

- シミュレータ
- テストベンチ（Verilog-HDLファイル）

シミュレータ

様々なシミュレータが出回っています。例えば、WikipediaでVerilogを検索すると有名なシミュレータが挙げられています。ここではVeritakの使用を前提に話を進めます。

テストベンチ

テストベンチは設計した回路(DUT)を検証するために必要な信号（DUTに与える信号）を生成したり検証を効率よく行うための結果表示などの方法を記述します。テストベンチは回路へ変換する必要が無いため分かりやすく記述する事、効率よく記述する事を優先して記述します。従って、回路設計では使用しない（できない）記述方法を使用する事が出来ます。

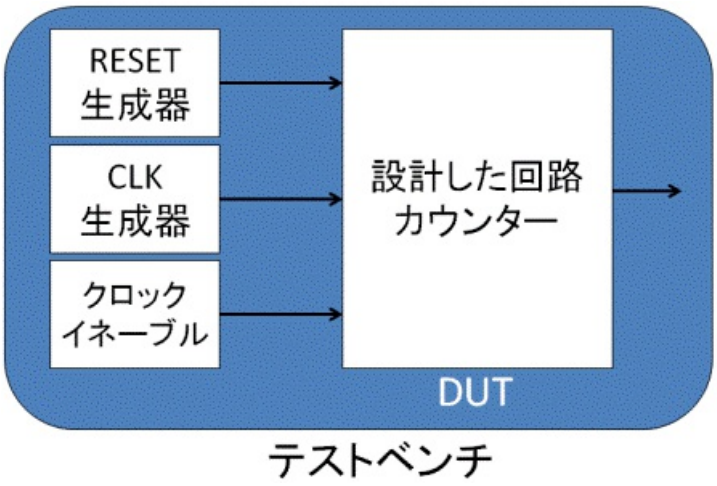
テストベンチ

FPGAの回路をHDLで記述し、その回路をシミュレーションして動作検証する場合を想定してみます。

テストベンチもVerilog-HDLモジュールです。他のモジュールと同様にテキストで記述します。

設計した回路はFPGAの回路でありPCB基板の一部です。基板上には発信器、リセットスイッチ、LEDや外部インターフェイス用コネクタが実装されています。設計した回路を動作させて検証するためには、それらの外部部品の動作を模して設計した回路に信号を与えなければいけません。同期回路の場合はクロックを与えなければいけませんし、初期状態を確定さえるリセット信号を与える必要があります。実際の動作を模すために信号を発生させるのがテストベンチです。

下にカウンター動作検証の為のテストベンチ例を挙げます。



設計したカウンタは入力として3つの入力：リセット、クロックとクロックイネーブル、を持っています。このカウンター回路をテストする為には先に挙げた3つの入力を与える必要があります。

テストベンチの記述方法に決まった方法はありません。例えば、外部メモリを使用しているので外部メモリを模す回路（モデル）をテストベンチに書かなければいけないか

と言うと、必ずしもその必要はありません。もちろん、完璧に動作するモデルが手に入るなら接続して検証する事も良いと思いますが、現実にはモデルも自ら製作する事になります。この事は開発期間、検証時間の増大を意味します。モデル動作の正当性の検証は特に重要です。モデルが間違っていたら検証結果も間違うからです。私達が設計する回路は企業が設計する様な複雑な回路は珍しいです。ですから、モデルをきちんと組み立てるのではなく、設計者が何を検証したいのか、どのように検証すれば動いているとみなす事が出来るのかを良く考え、テストしたい回路の入力を工夫して与える事で検証する事が重要になります。要求される開発期間を考慮するとシミュレーションにより完全に検証する事は不可能に近いです。

実際の信号を完全に模す環境を製作する事は開発期間を考慮すると不可能に近く長時間シミュレーションを行う事も難しいです。シミュレーションを行う時に大切な事をはテストベンチを工夫して作製して、最小の努力で出来るだけ大きな検証効果を上げる事が出来るテストベンチを製作することです。自分が設計した回路を十分に理解していればどのような信号が入力された時の動作が不安なのか、想定しない信号が入力された時にどのような動作なになるのか分かっているはずです。設計者が気になる部分を効率よく検証できるテストベンチを記述するようにしてください。

シミュレーション特有の記述

テストベンチはVerilog-HDLモジュールですが、他のモジュールと異なり回路へ変換する必要がありません。従って、回路化することを意識せずに記述することができます。文法で許されている記述をすべて使用する事が出来ます。

次の節から実際の例を挙げて説明します。

4.1 カウンターのシミュレーション

8bitカウンターを例にテストベンチについて説明します。

テストする回路

検証するカウンターは下の8bitカウンターです。

```
/* 8-bit conter */
module COUNTER(
    CLK         ,   // in   : System clock
    RST         ,   // in   : System reset
    CE          ,   // in   : Clock enable
    Q               // out  : Count[7:0]
);

//-------- Input/Output -------------
    input           CLK         ;
    input           RST         ;
    input           CE          ;

    output  [7:0]   Q           ;

//-------- 8-bit counter -------------
    reg     [7:0]   Q           ;

    always@ (posedge CLK) begin
        if(RST)begin
            Q[7:0]  <= 8'd0;
        end else begin
            if(CE)begin
                Q[7:0]  <= Q[7:0] + 8'd1;
            end
        end
    end

endmodule
```

このモジュールのポートは下の部分です。

```
    CLK         ,   // in   : System clock
    RST         ,   // in   : System reset
    CE          ,   // in   : Clock enable
    Q               // out  : Count[7:0]
```

CLK, RST, CEを与えないとこの回路は動きません。そこで、テストベンチでこれらの信号を生成する必要があります。

テストベンチ

必要最低限必要な機能に注意してテストベンチを書いてみました。

```
/*****************************************************************************
*                                                                            *
* カウンターのテストベンチです                                               *
*                                                                            *
*****************************************************************************/
`timescale 1ps/1ps

`include "COUNTER.V"

module COUNTER_TB;

    reg             OSC50M          ;
    reg             CLKENB          ;
    reg             RST             ;

    wire    [7:0]   Q               ;

//---------------------------------------------------------------------------
//  DUT
//---------------------------------------------------------------------------
    COUNTER     DUT(
    // System
        .CLK        (OSC50M         ),  // in   : System clock
        .RST        (RST            ),  // in   : System reset
        .CE         (CLKENB         ),  // in   : Clock enable
        .Q          (Q[7:0]         )   // out  : Count[7:0]
    );

//---------------------------------------------------------------------------
//  Clock generator
//---------------------------------------------------------------------------

    parameter   OSC50M_PERIOD   = 20000;    // ps

    initial begin
        OSC50M      = 1'b0;
    end

    always #(OSC50M_PERIOD/2) begin
        OSC50M  <= ~OSC50M;
    end

    reg     [1:0]   clkEnbCntr      ;

    initial begin
        CLKENB      = 1'b0;
        clkEnbCntr  = 2'b0;
    end

    always@ (posedge OSC50M) begin
        clkEnbCntr[1:0] <= clkEnbCntr[1:0] + 2'd1;
        CLKENB          <= (clkEnbCntr[1:0]==2'd1);
    end

//---------------------------------------------------------------------------
//  Reset generator
//---------------------------------------------------------------------------
    initial begin
        RST = 1;
        repeat(20) @(negedge OSC50M);
        RST = 0;
    end

//---------------------------------------------------------------------------
// Test vector
//---------------------------------------------------------------------------
    initial begin
        @(negedge OSC50M);
        while(RST) @(negedge OSC50M);
```

```
        repeat(100) @(negedge OSC50M);

        $finish;
    end

//---------------------------------------------------------------------------
endmodule
```

初めて登場した記述が幾つかあると思います。上から順に説明します。

環境設定

テストベンチでもモジュールの範囲はmoduleからendmoduleまでです。キーワードmoduleの前の文は環境設定です。この例ではシミュレーションを行う時間の単位、設計した回路の読み込みを指定しています。

```
`timescale 1ps/1ps
```

これはシミュレーションを行う時間単位を指定しています。`timescaleがキーワードです。シミュレーションは連続時間で行うのではなく離散時間で行います。設計した回路に合わせて設定します。キーワードに続く1ps/1psが単位時間と丸め精度を表しています。通常は同じ値に設定します。

Xilinx FPGA

Xilinx社から提供されている回路ライブラリは`timescale 1ns/1psが仮定されているものが多いようです。Xilinx FPGAを使用する時はライブラリを使用しなくても1ns/1ps設定が無難です。

```
`include "COUNTER.V"
```

設計した回路の読み出しです。シミュレータによっては`include文を使用しないでシミュレータのプロジェクトファイルで設定しないと誤動作するものがあります。自分の環境に合わせて設定してください。ここではVeritak使用を仮定して話を進めます。

モジュール宣言と内部信号宣言

```
module COUNTER_TB;

    reg             OSC50M          ;
    reg             CLKENB          ;
    reg             RST             ;

    wire    [7:0]   Q               ;
```

今まで説明した回路と同様テストベンチも一つのモジュールですのでmoduleからendmoduleまでがモジュール内動作の記述です。

設計する回路と異なりポートリストがありません。これは、開発した回路を動作させる環境を記述した回路なのでこのモジュールが最上層モジュールであり閉じていなければいけないからです。

次は内部信号の記述です。テストするカウンター回路に入力する信号はクロック、クロックイネーブル、リセットでした。それらの信号を与えるので記憶素子として宣言してあります。記憶素子として宣言する理由を説明します。信号を与えるには時間と値を指定します。例えば時間0ではクロックはL、10ns時にH、20ns時にLなど。変化点の時間と値を指定し変化点以外では値を保持するようにします。これは、値の設定方法が論理回路ではありませんが値を記憶する記憶素子です。直接値を設定できるテストベンチ用記憶素子だと考えください。最後にカウンターの出力信号の為のwire宣言があります。

テストする回路の組み込み

テストする回路の事をDevice Under Test(DUT)と呼びます。

組み込み方法は通常の回路設計時と同じです。

```
//---------------------------------------------------------------------------
//  DUT
//---------------------------------------------------------------------------
    COUNTER     DUT(
    // System
        .CLK        (OSC50M         ),  // in   : System clock
        .RST        (RST            ),  // in   : System reset
        .CE         (CLKENB         ),  // in   : Clock enable
        .Q          (Q[7:0]         )   // out  : Count[7:0]
    );
```

クロック生成回路

クロック生成回路には良く使われる記述方法があります。クロック生成は下の部分です。

```
//---------------------------------------------------------------------------
//  Clock generator
//---------------------------------------------------------------------------

    parameter   OSC50M_PERIOD   = 20000;    // ps

    initial begin
        OSC50M = 1'b0;
    end

    always #(OSC50M_PERIOD/2) begin
        OSC50M  <= ~OSC50M;
    end
```

クロック周期の設定

```
    parameter   OSC50M_PERIOD   = 20000;    // ps
```

キーワードparameterは定数宣言です。OSC50M_PERIODは20000と定義されています。シミュレーション環境は1psを時間単位と設定さているので20nsに相当します。

下のalways文で使用する時に直接数字を使用しても良いのですがコードの読みやすさを考慮して定数を使用しています。この様にすると50M発振器の周期だとすぐにわかります。

クロック信号の初期化

```
    initial begin
        OSC50M = 1'b0;
    end
```

initial文で初期化しています。initial文はシミュレータ実行時に一度だけ実行される命令でテストベンチで良く使用されます。ここでは''OSC50M = 1'b0;''の一行だけですので、シミュレーション開始時に0に設定されます。最初0に設定されたOSC50Mは記憶素子出力なので何もしなければこの後は0のままです。

回路合成時の注意

initial文を回路記述に使用しないでください。使用しなくても問題ないはずですが、どうしても使用したい時は合成ツールのマニュアルを良く読んでから使用するようにしてください。

クロック生成

```
    always #(OSC50M_PERIOD/2) begin
        OSC50M  <= ~OSC50M;
    end
```

このalwaysは以前登場したものと少し違います。以前@だった文字が#へ変わっています。

@は条件式が成立した時に実行しますが、#は#の次に書かれている値だけ待ちます、ここではOSC50M_PERIOD/2。#の値だけ待つことを毎回alwaysで実行するので10ns毎に実行します。実行する内容はOSC50Mの出力を反転させた値を次の出力に設定です。

結果として初期値0、その後は10ns毎に値を反転させる動作になりますのでクロックが生成された事になります。

クロック・イネーブル生成回路

クロックイネーブルの回路としてOSC50Mで数えて4クロック毎に1クロック周期Hになる回路を作りました。

```
    reg     [1:0]   clkEnbCntr      ;

    initial begin
        CLKENB      = 1'b0;
        clkEnbCntr  = 2'b0;
    end

    always@ (posedge OSC50M) begin
        clkEnbCntr[1:0] <= clkEnbCntr[1:0] + 2'd1;
        CLKENB          <= (clkEnbCntr[1:0]==2'd1);
    end
```

クロック生成の時と同じです。

最初にクロックイネーブルを生成する為の分周用カウンター信号を宣言しています。その後initial文で初期化を行い、通常の回路と同じ記述方法でalways文を使用しています。クロック生成回路でも書きましたが、設計するモジュールの回路記述でinitial文で初期化しalways文で回路動作記述する方法は使わないでください。

リセット生成回路

リセット生成をテストベクタの中に記述する事も出来ますが、ここではテストベクタと分けて独立に記述してみました。

```
//---------------------------------------------------------------------------
//  Reset generator
//---------------------------------------------------------------------------
    initial begin
        RST = 1;
        repeat(20) @(negedge OSC50M);
        RST = 0;
    end
```

initial文ですから、シミュレータ起動時に文中の命令を上から下へ一度だけ実行します。

最初RST = 1;によりRST信号はHに設定されます。

```
repeat(20) @(negedge OSC50M);
```

キーワードrepeat(20)は繰り返し命令です。repeat()の次の命令@(negedge OSC50M)を括弧中の数字20回繰り返します。この分は@(negedge OSC50M)を20回繰り返します。

@(negedge OSC50M)はOSC50Mの立下り時のみ動作する条件ですから、この文はOSC50Mの立下りを20回見つけるまで繰り返す、すなわち、OSC50Mの20クロックだけ待つ、命令になります。

次にRST = 0でRST信号がLに設定されています。

以上の記述でシミュレータ起動時からOSC50Mで20クロック時間リセットをかける回路を模しています。

テストベクタ

```
//---------------------------------------------------------------------------
// Test vector
//---------------------------------------------------------------------------
    initial begin
        @(negedge OSC50M);
        while(RST) @(negedge OSC50M);

        repeat(100) @(negedge OSC50M);

        $finish;
   end

//---------------------------------------------------------------------------
endmodule
```

今までの説明に登場していないのはwhileと$finishを説明します。

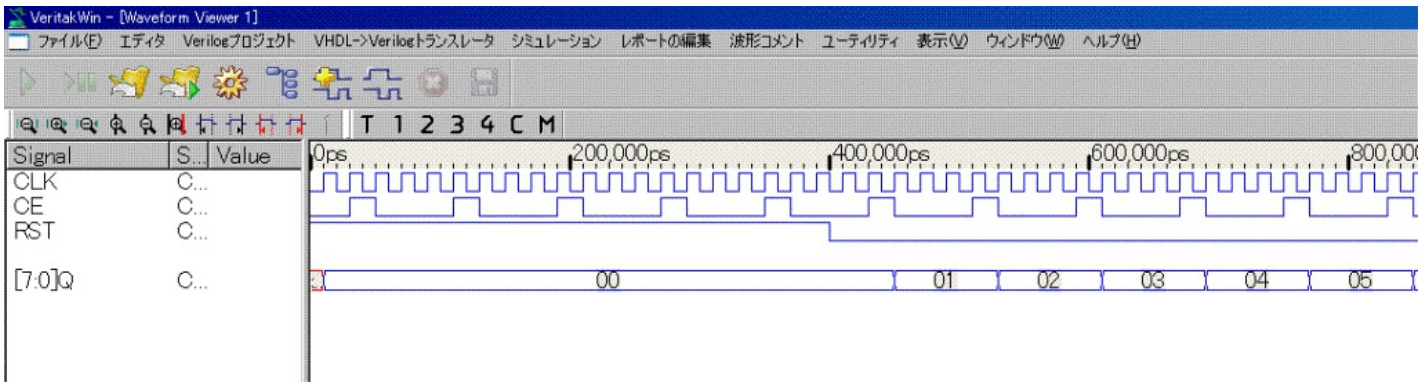
whileは括弧の中の条件式が成立している間、次に書かれている命令@(negedge OSC50M)を実行し続けます。RST=Hの時は立下りエッヂを待ち続けるのでRST=Lになった直後の立下りエッヂで次の行を実行します。

$finishはシミュレーション終了命令です。Veritakの場合、この命令が無いと走り続けます、止まりません。

テストベクタの動作をまとめます。シミュレータが起動するとシステムクロックの立下りを待ちます。その後リセットが解除されるまで待ち、100クロック時間後にシミュレーションを停止します。

シミュレーション結果

Veritakでのシミュレーション結果は下の様になり、正しくカウンタ動作している事が分かります。



目次へ戻る

技術教育のページへ戻る